ECHO: Instantaneous In Situ Race Detection in the IDE

Sheng Zhan, Jeff Huang
Parasol Laboratory
Texas A&M University, USA
echo@tamu.edu, jeff@cse.tamu.edu

ABSTRACT
We present ECHO, a new technique that detects data races instantaneously in the IDE while developers code. ECHO is the first technique of its kind for incremental race detection supporting both code addition and deletion in the IDE. Unlike conventional static race detectors, ECHO warns developers of potential data races immediately as they are introduced into the program. The core underpinning ECHO is a set of new change-aware static analyses based on a novel static happens-before graph that, given a program change, efficiently compute the change-relevant information without re-analyzing the whole program. Our evaluation within a Java environment on both popular benchmarks and real-world applications shows promising results: for each code addition, or deletion, ECHO can instantly pinpoint all the races in a few milliseconds on average, three to four orders of magnitude faster than a conventional whole-program race detector with the same precision.

CCS Concepts
• Software and its engineering → Software maintenance tools; Software verification and validation;

Keywords
Data Races, Change-aware, Instantaneous Detection, IDE

1. INTRODUCTION
Data races are among the hardest to debug types of bugs in software systems. As software becomes more parallel, race detection techniques are proliferating [1, 2, 3, 4, 5, 6]. Several industrial-strength tools [7, 8, 9] have also been deployed. Most techniques and tools, however, are designed for late phases of the software development cycle, e.g., testing or production, where the whole program is completed. Although races detected in a later phase are more likely to be real bugs, scaling to programs with a large code base without sacrificing detection coverage or accuracy is difficult. Moreover, the later a bug is found, the more expensive it would be to fix it [10].

We advocate detecting races early in the programming phase (ideally, in the IDE) such that it is both easier to scale the race detector, by amortizing the analysis cost, and cheaper to fix the detected races, by providing developers early feedback. However, existing IDEs (e.g., Eclipse [11]) lack the support for detecting sophisticated bugs such as data races, because of the expensive analysis cost. For example, static race detectors [1, 2, 12] typically require pointer analysis, which often takes several seconds or minutes to compute for realistic programs. Upon a code change in the IDE, instead of running a conventional race detector and waiting for seconds or minutes, developers would favor an in situ race checker running in the background that, similar to checking syntax errors, detects races “instantaneously” as they are introduced, and as non-intrusively as possible.

In this paper we present ECHO, a new technique and a prototype tool that realize the above vision in Eclipse. A snapshot of ECHO is shown in Figure 1 (see also a video demo at [13]). The two statements at lines (20, 37) form a race on a shared variable x. ECHO detects this race and displays the bug warning instantly (i.e., in a few milliseconds) as the two statements are introduced into the program. When either of the two statements is deleted or the statement at line 37 is moved into the synchronized region, ECHO will invalidate the warning, again, instantly.

Figure 1: Instantaneous race detection by ECHO.
In a nutshell, ECHO leverages the fact that program-
ing often involves frequent but small changes, which can
be analyzed quickly together with only their respective de-
pendencies without re-analyzing the whole program. Yet,
for race detection, the problem of how to efficiently up-
date the change effects and correctly relate them to races
is quite challenging. We develop a new change-aware race
detection algorithm based on a novel graph representation
of the happens-before relation that handles a realistic subset
of multithreaded Java programs (see Section 3), supporting
both addition and deletion of different types of statements.

A critical component of our algorithm is an on-the-fly
points-to analysis that determines the heap locations ac-
cessed by program statements and pointer aliases for rea-
soning about lock operations. Although points-to analysis
has been intensively studied before [14, 15, 16] including
a few incremental algorithms [17, 18], there is no previous
 technique that is applicable within an IDE in which both
code addition and deletion must be handled. In particular,
handling deletion is difficult because it may involve complex
data-flow analysis and invalidation of the existing points-to
set. A reset-then-recover algorithm does not scale because
the analysis is cubic in the program size. We develop a novel
reachability-based algorithm that optimizes the invalidation
of the points-to set when a statement is deleted, achieving
as much as 41X (see Section 4.1) speedup over the reset-
then-recover algorithm on a real-world application.

Like other static race detectors, ECHO is incomplete and
can report false positives due to the limitation of static
analysis. However, compared to conventional race detec-
tors, we argue that ECHO is less over-whelming to de-
velopers as they receive immediate feedback on potential
races rather than getting a large number of warnings all
at once. Moreover, ECHO implements two optimizations
to improve precision. First, ECHO uses a hybrid algo-
rithm combining happens-before and lockset. As observed
by other researchers [19, 1], the hybrid algorithm is effec-
tive in pruning false positives reported by purely lockset-
based detectors. Second, ECHO builds on top of an object
field-sensitive (but context-insensitive), locally flow-sensitive
Andersen-style analysis [20]. The field and locally flow sen-
sitivity effectively reduces false positives caused by object-
level false sharing and flow-insensitivity within a method.

Our evaluation on a variety of popular benchmarks and
real-world applications shows that ECHO detects 100% of
the known races with a 36% false positive ratio and it takes
only 1-5ms on average to handle each change. Compared
to a whole-program race detector with the same recall and
only 1-5ms on average to handle each change. Compared
the known races with a 36% false positive ratio and it takes
real-world applications shows that
sensitivity e
Andersen-style analysis [20]. The field and locally flow sen-
sitive in pruning false positives reported by purely lockset-
rithm combining happens-before and lockset. As observed
to improve precision. First,

2. OVERVIEW

We first present an overview of ECHO with an artificial
example, and then discuss the technical challenges.

Example. Imagine that in an IDE the developer has writ-
ten the Java program in Figure 2(a) but not yet the code
in the gray region (i.e., the changes ①–③). The program
starts two threads testing a Vector container by storing and
retrieving objects of the Conference class, which has two at-
tributes, name and year. The main thread (T1) first creates
two Conference objects, c1 and c2, and assigns c1 to another
object reference c2. It then adds c2 to the vector v and
starts the child thread (T2) passing v as an argument. T2
traverses the vector and prints out each element contained
in it. The Vector implementation here is not thread-safe,
because its methods are not synchronized. However, there
is no data race in this program so far, because all operations
are ordered by happens-before, i.e., T2 must execute after
the thread start operation by T1 at line 39.

2.1 ECHO in Action

Suppose the developer now performs the changes ①–③.
We next show how ECHO reacts to them one by one.

Change ①. When the first change ① v.add(c3) is in-
roduced, ECHO displays two races between lines (19,23)
and (23,27). The reason is that the change adds the second
Conference object (referenced by c3) to the vector, which
modifies both the size of the vector (count) and the corre-
sponding array element (elems) at line 23. These two writes
are not ordered with the two method calls v2.size() and
v2.get(i) by T2, which respectively read count and elems
on the same vector at lines 19 and 27. None of these four
accesses is protected by any lock and they form two races.

Changes ②③. Upon seeing the two race errors, the devel-
oper attempts to fix them by introducing Change ②: adding
synchronized keyword to both add(a) and size(). ECHO
detects that the race (19,23) is fixed (because both of the two
accesses are now protected by the same lock) and clears the
warning. However, the other race (23, 27) remains because
the access at line 27 is not protected. As a result, the devel-
oper proceeds to introduce Change ③: adding synchronized
to get(i). After this change, the race warning (23,27) also
disappears because ECHO detects that both accesses to the
array element are now protected by the same lock.

Changes ④⑤. Now the developer adds ④ c3.increment-
Year() at line 41. ECHO detects a new race (8,10), because
this method call modifies the attribute year at line 8 and it
is not ordered with the method call p.toString() by T2 at
line 51, which reads year at line 10. Both c3 and p can refer
to the second Conference object and these two methods are
not synchronized, so the two accesses to year form a real
race. To fix this race, instead of adding synchronization,
the developer realizes that c3 should not be added to the
vector and hence performs Change ⑤: deleting v.add(c3)
at line 40. Upon the deletion, ECHO invalidates the race
warning (8,10) because now lines 8 and 10 access different
objects and p cannot refer to the second Conference object.

2.2 ECHO in a Nutshell

Figure 2(b) shows an architectural overview of ECHO,
consisting of three components: a change tracker, a race
detection engine, and a race display器. The first and the third
components are both IDE-specific. The second compo-
nent takes one or more changes as input and runs a change-
aware algorithm to detect races. The algorithm relies on three mutually dependent graphs – a points-to graph, a call graph, and a static-happens-before (SHB) graph, all of which are computed in a change-aware manner: only those facts (nodes and edges) in the graph that are affected by the change are recomputed and the rest of the graph remains the same. The points-to graph and call graph are standard. We refer the readers to previous work [14, 21] for their background. The SHB graph is a new data structure.

**Static Happens-Before (SHB) Graph** The SHB graph augments the call graph with directed edges representing the happens-before relation between abstract threads and heap accesses. However, in the SHB graph, the change is recomputed and the rest of the graph remains the same. The points-to graph and call graph are standard. We refer the readers to previous work [14, 21] for their background. The SHB graph is a new data structure.

**Algorithm 1 ECHO Race Detection (Δₚ)**

1. **Input:** Δₚ - a set of program changes.
   - Additions: +{a₁, a₂, ...}; deletions: −{d₁, d₂, ...}.
2. **Global states:** ptg - points-to graph; shb - static-happens-before graph.
3. **Δ₂ptg, Δ₂shb** - UpdatePointsToAndCallGraph(Δₚ);
4. **Δₚshb** - UpdateSHBGraph(Δₚ, Δ₂ptg, Δ₂shb);
5. DetectDataRaces(Δ₂shb, Δ₂ptg).

**Algorithm Overview** An overview of our race detection algorithm is shown in Algorithm 1. Given a set of program changes Δₚ (including both addition and deletion¹), the points-to graph and call graph are updated first (we will show how in Section 3). Then, Δₚ together with the changes in the call graph (Δ₂ptg) and the SHB graph (Δ₂shb) can lead to new races or invalidate any existing races. There are two basic steps: finding and invalidating conflicting accesses and checking happens-before and lockset. The first step tracks changes of states (i.e., read and write sets) associated with each AHL. If the write set contains accesses from at least two different abstract threads, or at least one access from a thread that is different from any thread in the read set, the two corresponding accesses are considered conflicting. The second step checks for the two conflicting accesses their happens-before relation and locksets, which can be computed using the SHB graph. If the two accesses are not ordered by happens-before or their locksets do not overlap, they are reported as a race.

**Example** The points-to graph and the SHB graph before the changes 1-5 are shown in Figure 2, ignoring the colored nodes and edges. When 1 v.add(c3) is added, both of the two graphs are updated. A new edge c₃ → e is added in the points-to graph because c₃ is passed as the method argument. A new node v.add is added in the SHB graph and a new edge from t.start is added to this new node. In addition, because p can now refer to o₃₅, the read set of o₃₅.year is updated to include T2. The new v.add node has the same read and write statements as that of the first v.add node before t.start. However, the difference is that this node is not ordered with the nodes from T2. Hence, by checking happens-before and lockset related to the new accesses to o₃₅.t[x] and o₃₅.count, our algorithm detects two races, (19,23) and (23,27). Similarly, when 4 c₃.incrementYear() is added, a new c₃.incrementYear node is added to the SHB graph and a new edge from t.add is added to the new node. Both the read and write sets of o₃₅.year are updated to include T1, because c₃.incrementYear() reads and writes the field.

¹Note that a code update can be treated as two changes: deletion of the old statement, and addition of the new statement. Large code chunks can be treated as a sequence of small changes.

---

**Figure 2: ECHO Technical Overview.**

- **(a) Example**
- **(b) Architecture Overview**
- **(c) Points-to Graph**
- **(d) Static Happens-Before Graph**
P ::= defn * e  \hspace{1cm} \text{(program)}
defn ::= class C {field * meth * }  \hspace{1cm} \text{(class decl)}
field ::= T f  \hspace{1cm} \text{(field decl)}
meth ::= C m(args) {s * return z }  \hspace{1cm} \text{(method decl)}
s ::= 
1. x = new C \hspace{1cm} \text{(allocate)}
2. x = y \hspace{1cm} \text{(simple assign)}
3. x = y, f \hspace{1cm} \text{(field read)}
4. x, f = y \hspace{1cm} \text{(field write)}
5. x = o, m(args) \hspace{1cm} \text{(method call)}
6. x = y[i] \hspace{1cm} \text{(array read)}
7. x[i] = y \hspace{1cm} \text{(array write)}
8. t.start() \hspace{1cm} \text{(thread fork)}
9. t.join() \hspace{1cm} \text{(thread join)}
10. synchronized(x) {s * } \hspace{1cm} \text{(lock)}
11. loop(b) {s * } \hspace{1cm} \text{(loop)}
\hspace{0.5cm} | \hspace{0.5cm} if(b) {s * } \hspace{1cm} \text{(conditional)}
\hspace{0.5cm} | \hspace{0.5cm} e \hspace{1cm} \text{(other)}

Figure 3: SIMJava.

year, and c3 can refer to o35. The read set of \( o35.year \) contains T2, so the two accesses at lines 8 and 10 are conflicting. By checking their happens-before and locksets, our algorithm detects a new race (8.10).

2.3 Technical Challenges

To achieve both fast speed and good precision, there are several tough technical problems that we must solve:

1. **Change-aware race detection.** How to correctly and efficiently react to a program change? How to correctly handle different types of changes? How to efficiently maintain the happens-before relation and lockset upon a change?

2. **Change-aware points-to analysis.** How to soundly update the points-to graph and call graph upon a change? By soundness, we mean that any true points-to relation must be represented in the points-to graph. Meanwhile, we would like to compute a points-to graph that is as precise as possible. For example, in Figure 2, before Change \( \odot \), the variable \( p \) cannot refer to \( o35 \). Otherwise, a false positive would be reported.

3. **Sound static happens-before graph.** How to construct a sound SHB graph such that any true happens-before relation is represented and no reachable heap access is missed? How to handle back edges caused by loops or recursion? How to identify abstract threads that may have multiple runtime instances?

We next present our algorithms to address these challenges.

3. ALGORITHM

We first introduce a multithreaded language SIMJava, which contains a subset of JAVA basic constructs for multithreaded programming. Based on SIMJava, we then present our change-aware analysis algorithms.

The SIMJava Language. SIMJava is inspired from ConcurrentJava [22], with a few differences and extensions that make it more powerful for expressing concurrency and also cleaner for change-aware analysis. Figure 3 shows the syntax. SIMJava supports three types of inter-thread synchronization operations, fork, join, and synchronized. The fork and join operations together form the inter-thread happens-before relation and synchronized forms the lock mutual exclusive relation. Statements \( \odot \& \odot \) include the typical operations for object allocation, assignments, field and array read and write, method invocation, as well as the concurrency primitives. All these statements are analyzed in our race detection. In addition, SIMJava supports the loop operation \( \odot \text{loop}(b)\{s *\} \) that evaluates a boolean variable \( b \) and iterates the statements \( s * \). The loop operation reflects loop constructs, such as for and while. For race detection, loop operations must be considered in our algorithm because they may spawn multiple threads and may also introduce multiple other types of synchronizations. However, conditions are ignored because our algorithm is path-insensitive.

3.1 Change-aware Data Race Detection

The core of our algorithm is a change-aware static happens-before (SHB) graph powered by an on-the-fly points-to analysis. We first present the SHB graph construction algorithm.

3.1.1 SHB Graph Construction

Starting from a unique entry method (e.g. main), the SHB graph is constructed following the rules in Table 1. There are nine different types of nodes in the SHB graph, corresponding to the nine statements \( \odot \& \odot \) in SIMJava. \( \odot \text{(field read)} \) and \( \odot \text{(field write)} \) are similar to \( \odot \text{(array read)} \) and \( \odot \text{(array write)} \), except that the index to field is fixed and \( \odot \text{(other)} \) are omitted.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \odot x = y.f )</td>
<td>read(y.f)</td>
</tr>
<tr>
<td>( \odot x.f = y )</td>
<td>write(x.f)</td>
</tr>
<tr>
<td>( \odot x = o.m(y') )</td>
<td>( \forall O_c \in pts(o) : \text{call}(O_c,m) )</td>
</tr>
<tr>
<td>( \odot t.start() )</td>
<td>( \forall O_c \in pts(t) : \text{fork}(O_c) )</td>
</tr>
<tr>
<td>( \odot t.join() )</td>
<td>( \forall O_c \in pts(t) : \text{join}(O_c) )</td>
</tr>
<tr>
<td>( \odot \text{synchronized}(x){s *} )</td>
<td>( \forall O_c \in pts(x) : \text{lock}(O_c) )</td>
</tr>
<tr>
<td>( \odot \text{loop}(b){s *} )</td>
<td>unroll twice: ( s * )</td>
</tr>
</tbody>
</table>

Two fake nodes: \( \forall O_c \in pts(t) : \text{start}(O_c) & \text{end}(O_c) \)

Edges: \( \forall O_c \in pts(t) : \text{fork}(O_c) \rightarrow \text{end}(O_c) \)
\( \text{end}(O_c) \rightarrow \text{join}(O_c) \)

Method call \( s (o,m) : \forall O_c \in pts(o) : \Rightarrow \text{Node}(s) \rightarrow \text{FirstNode}(O_c,m) \)
\( \text{LastNode}(O_c,m) \rightarrow \text{NextNode}(s) \)
\( \forall s_1,s_2 \in m & s_1 < s_2 \Rightarrow \text{Node}(s_1) \rightarrow \text{Node}(s_2) \)

Table 1: Static HB Graph Construction. Array accesses are treated similarly to field accesses (as to a single field) and are omitted.
read/write node, we maintain a link from the node to one
(or more) AHL, which corresponds to the abstract data the
node accesses. The AHL is identified by \(O.f\) (for field) or
\(O.x\) (for array), where \(O\) is an abstract object in the points-to
set of the base variable. Each AHL is associated with
two states: a read set and a write set, recording write and
read accesses to the location from abstract threads. This
information is used to identify conflicting heap accesses.

**Abstract Thread and Lock.** For the other nodes, each
node is associated with one (or more) abstract object, which
is computed using the points-to set of the corresponding
base variable. For example, for fork and join (and start
and end), their abstract object is the corresponding abstract
node, identified by the points-to set of \(t\) in \(t.start()\),
\(t.join()\), or the main method (for only the main thread).
For lock and unlock, their abstract object is the correspond-
ing abstract lock: the points-to set of \(x\) in \(\text{synchronized}(x)\).

**Handling Loop.** For loop statements \(\Theta (\text{loop}(b)(s^{*}))\),
we create more than one sequence of nodes for \(s^{*}\), because
each \(s\) may generate multiple reads/writes or fork multiple
threads. The challenge is that the number of loop iterations
is unknown statically. Nevertheless, for race detection, it
suffices to unroll the loop twice. The reason is that data
races involve only two abstract threads and two memory ac-
cesses. Unrolling a loop twice will guarantee to expose the
same set of races as unrolling more than two times. Similar-
ly, we handle recursion by unrolling all loops in the call
graph twice and removing the corresponding back edges.

**Statement Location.** Each node in the SHB graph is
also associated with a unique location, corresponding to the
program location of the statement. The unique location is
used to determine the program order for statements from the
same method. For synchronized blocks and loop statements,
the locations of their corresponding nodes are treated in the
following way. For lock and unlock, their locations corre-

dent to the locations of "(" and ")" of the synchronized
block. For loops, we add a loop iteration identifier (either
11 or 12) to each node, all nodes with 11 should happen be-
fore nodes with 12 unrolled from the same loop statement.
Together with the call graph, the node location information
is used to compute the intra-thread happens-before relation.

**Happens-Before.** The happens-before edges are con-
structed over fork→start and end→join for each abstract
thread object and over method calls. For a method call \(s, \ o.m\), for each abstract object, \(O.c\), in the points-to set of \(o\), an
edge is added from its corresponding node. Node(s), to the
first node of the callee method. FirstNode(O.c,m). In ad-
nition, an edge is added from the last of the callee method,
LastNode(O.c,m), to the next node of \(s\), NextNode(s).
Furthermore, happens-before edges are added between con-
secutive nodes in \(O.c,m\) following the program order.

**Lockset.** lock and unlock nodes do not introduce happens-
before edges. Instead, we associate every memory access
node (read and write) protected by each pair of lock and
unlock nodes with a lockset and add all the abstract lock
objects to which these lock variables may refer to the lock-
set. The lockset is used together with happens-before to
improve precision of race detection.

### 3.1.2 Change-aware SHB Algorithm

If any of these nine types of statements is added or deleted,
our change-aware SHB algorithm updates the SHB graph.
This step is relatively straightforward by following the rules
in Table 1. For addition, we first insert the corresponding
nodes (introduced by the new statement) into the SHB
graph according to the statement location. We then add
the links for AHL and add the happens-before edges according
to the points-to set. For deletion, we simply remove all the cor-
responding nodes and their links and edges from the graph.
If a removed node \(n\) is between two nodes \(n_1 \rightarrow n \rightarrow n_2\),
then the two nodes will be connected \(n_1 \rightarrow n_2\). For loop
statements, their addition and deletion are equivalent to
adding and deleting \(s^{*}\) in the loop body. For synchronization
statements, we update the lockset of each read and write
node that they protect accordingly.

For statements \(\Theta\) (allocate) and \(\Theta\) (simple assignment),
they may change the points-to graph and call graph. For
points-to changes, we update the SHB graph by adding or
deleting the corresponding nodes/edges/links according to
the changed points-to set of each base variable. For call

graph changes, we delete only the related edges but not the
nodes, to reuse the nodes later if a method call statement
to the same method is added.

A caveat is that a statement may appear as multiple nodes
in the SHB graph because the statement is in a loop or its
enclosing method is called in multiple places. Therefore, for
changes to these statements, we must track and update all
their occurrences in the SHB graph. We track these state-
ments by maintaining a map from each method to its loca-
tions in the graph and a boolean state for each statement
indicating if it is in a loop. For a statement change, we lo-
cate all their occurrences by checking both the map with its
enclosing method, and the boolean state. We do not han-
dle method recursion separately because recursion is already
handled by unrolling the loops twice in the call graph.

### 3.1.3 Change-aware Race Checking

The race checking procedure is triggered upon a change in
the SHB graph. There are three types of changes: links to
AHL, lockset, and happens-before. When a link to an AHL
is added or deleted, it means that a read or write node, \(X\), is
added or deleted, and we perform race checking specific to \(X\).
We first find all the pairs of conflicting nodes including \(X\) by
checking the associated read and write sets. Because any of
these pairs may become a race (or no longer a race), for each
pair, we check the happens-before relation and the lockset
condition between the two nodes. If the two nodes cannot
reach each other on the SHB graph and their locksets do not
intersect, we flag them as a race. The lockset condition here
is essentially a may-alias analysis that determines if two lock
variables may refer to a common lock. If the flag of any pair
is changed, we update the race warning in the IDE.

For happens-before changes, we only handle inter-thread
changes because intra-thread happens-before is determined
by program order and it alone cannot introduce new races or
invalidate existing races. There are two types of inter-thread
happens-before edges: (1) fork→start and (2) end→join.
For (1), we check only the conflicting node pairs involving
those nodes that happen before the fork node and those that
happen after the start node because only (the happens-
before relation of) those nodes can be affected by this happens-
before edge. Similarly, for (2), we check only the conflicting
data pairs involving those nodes that happen before the end
node and after the join node. For lockset changes, similarly,
we only find and check those conflicting node pairs involving
nodes whose locksets are changed.
3.2 Change-aware Points-to Analysis

Our change-aware points-to analysis builds on an on-the-fly Andersen-style algorithm [20]. It is context-insensitive, but field-sensitive and locally flow-sensitive, i.e., flow-sensitive within each method. The key novelty of our new algorithm (Algorithm 2) is to make the analysis more efficient in handling program changes including both addition and deletion.

**On-the-fly Andersen’s Algorithm.** Let $\text{pts}(v)$ denote the points-to set of a variable $v$ and $O_v$ the abstract object directly assigned to $v$. Points-to analysis is often cast as a graph closure problem. Each node represents a variable $v$ and has an associated points-to set $\text{pts}(v)$ or $O_v$. In Andersen’s algorithm, edges represent subset constraints between nodes: an edge $a \leadsto b$ means that $\text{pts}(a)$ is a subset of $\text{pts}(b)$.

For SIMJAVA, there are seven types of statements (1-7) relevant to points-to analysis. 1 (allocate) and 2 (simple assignment) are used to initialize the points-to graph, and the rest five (3-7) may add more edges on-the-fly. Table 2 shows an extended Andersen’s algorithm. The statement 6 (method call) is also directly related to call graph construction. When a new call graph edge is discovered, the points-to graph may also be updated because of the new points-to facts introduced by parameter passing and value returning. As a result, the on-the-fly algorithm works in a loop until reaching a fixed point, i.e., both the points-to graph and call graph are unchanged. In each iteration, a worklist is used to track the new points-to facts and the points-to information is propagated along the two graphs following the constraint rules in the second column of Table 2.

### 3.2.1 Statement Addition

Handling statement addition follows the same rationale as the on-the-fly Andersen’s algorithm. New points-to facts (nodes/edges) are first extracted from the added statement and put into the worklist. Then, the points-to information is computed along the relevant paths in the two graphs until reaching a fixed point. The key advantage of our new algorithm is that only those nodes in the paths related to the new facts are recomputed, all the other nodes are untouched.

The algorithm (Algorithm 2 lines 7-19) takes an added statement $s$ and its enclosing method $C.m$ as input ($C.m$ is needed for building the call graph). It first finds out all the new edges that $s$ may introduce using the function FIND-EDGES (Algorithm 3). FIND-EDGES handles each type of statements following the rules in the third column of Table 2. For the first two types (1 and 2), the new edges can be added straightforwardly (note that the side effect of Statement 1 ($x = y$) is handled by the fixed-point computation in Algorithm 2 at lines 9-19). For the other three (3-7), which we call complex statements, their corresponding edges are not fixed but depend on the points-to set of their base variable. For example, for 3 ($x = y.f$), suppose $\text{pts}(y)$ contains two objects $o_1$ and $o_2$, then two edges must be added: $o_1.f \leadsto x$ and $o_2.f \leadsto x$. Moreover, when $\text{pts}(y)$ is changed during the computation, the corresponding statement must be re-evaluated because new edges may be added or deleted. Therefore, in addition to finding edges for these complex statements, FIND-EDGES also maintains a map, $CS$, that records the corresponding complex statements and their methods for each base variable. For example, for 3 ($x = y.f$), $x \leadsto (C.m, s)$ is added to $CS$ for statement addition and deleted for deletion.

Statement 6 ($x = m.\text{method}(y')$) may additionally update the call graph, trigger new statements (in the callee method) to be added, or introduce points-to edges related to both formal parameters and return. We handle all these cases correspondingly in Algorithm 3.

For each new edge $src \leadsto dst$, if it is not already in the points-to graph, it is added to the graph and its points-to set is updated following the subset constraints: $\text{pts}(dst) \leftarrow \text{pts}(dst) \cup \text{pts}(src)$. If $\text{pts}(dst)$ is changed, all edges from $dst$ in the points-to graph are added to the worklist and re-processed to update the points-to set of $v$. In addition, all the complex statements that have $dst$ as their base variable will be re-processed because new points-to edges may be introduced by the change in $\text{pts}(dst)$.

### 3.2.2 Statement Deletion

Handling statement deletion is more complicated than addition. Intuitively, it is the reverse of addition and, if we can track the state changes of the points-to graph by each addition, we may undo the changes for deletion of the same statement. Yet, this intuition is not true because points-to analysis is not “reversible”. Deletion is fundamentally different from insertion in that it requires updating not only the pointer information of the specific change, but also previous changes that are dependent on this change. Moreover, for large graphs, it is expensive to memorize the state changes.

**Reset-then-recover.** One (less efficient) solution (Algorithm 2 lines 20-40) is to reset the points-to sets of all relevant nodes and then recompute them. We can first find all the points-to edges that are related to the deleted statement, remove all these edges, and reset (set to empty) the points-to sets of their destination nodes as well as all nodes that they can reach. Then, following the same method for addition, we can add all the edges in the remaining points-to graph that can reach the reset nodes into the worklist and repeat the fixed point computation. This method is inefficient because the points-to sets of some reset nodes may in fact remain unchanged before and after the deletion, such that all the reset-then-recover computations are wasted.

**Reachability-based Algorithm.** Our optimized solution is based on two observations. First, after deleting an edge $x \leadsto y$, the points-to set of $y$ may remain the same if $x$ is still reachable to $y$. Second, for any abstract object $O$, if $O$ is reachable to $y$ then $O$ must be included in the points-
Algorithm 2 Change-Aware Points-to Analysis
1: Input: \( s \) – a new added or deleted statement in \( C.m \).
2: Global States: \( ptg \leftarrow \{ V_p, E_p \} \) – points-to-graph;
3: \( cg \leftarrow V_c, E_c \) – call graph;
4: \( pts \) – points-to-set function;
5: \( W \) – worklist;
6: \( CS \) – variable to complex statements;
7: 
8: \( W \leftarrow FINDEDGES(C.m, s) \);
9: while \( W \neq \emptyset \) do
10: \( e \leftarrow \text{SELECT FROM } W[/e] : src \sim \text{ dst} \)
11: if \( e \notin E_p \) then // \( e \) is a new points-to-edge
12: \( E_p \leftarrow E_p \cup e \);
13: \( pts(dst) \leftarrow pts(dst) \cup pts(src) \);
14: if \( pts(dst) \) changed then
15: \( \text{foreach } (\text{ dst } \sim v) \in E_p \)
16: \( W \leftarrow W \cup (\text{ dst } \sim v) \);
17: // process complex statements
18: \( \text{foreach } (C.m, s) \in CS(dst) \)
19: \( W \leftarrow W \cup FINDEDGES(C.m, s) \);
20: 
21: // Deletion - Reset-then-recover
22: \( W \leftarrow \emptyset \);
23: while \( W \neq \emptyset \) do
24: \( e \leftarrow \text{SELECT FROM } W[/e] : src \sim \text{ dst} \)
25: \( E_p \leftarrow E_p \setminus e \);
26: \( W \leftarrow \text{ Reset } \cup \text{ dst} \);
27: while \( W \neq \emptyset \) do
28: \( v \leftarrow \text{SELECT FROM Reset} \)
29: \( pts(v) \leftarrow \emptyset \);
30: \( \text{foreach } (v \sim \text{ dst}) \in E_p \)
31: \( \text{Reset } \leftarrow \text{ Reset } \cup \text{ dst} \);
32: \( \text{foreach } (\text{ src } \sim v) \in E_p \)
33: \( W \leftarrow W \cup (\text{ src } \sim v) \);
34: while \( W \neq \emptyset \) do
35: \( e \leftarrow \text{SELECT FROM } W[/e] : src \sim \text{ dst} \)
36: \( pts(dst) \leftarrow pts(dst) \cup pts(src) \);
37: if \( pts(dst) \) changed then
38: // process complex statements
39: \( \text{foreach } (C.m, s) \in CS(dst) \)
40: \( W \leftarrow W \cup FINDEDGES(C.m, s) \);
41: 
42: // Deletion - Reachability-based
43: \( W \leftarrow FINDEDGES(C.m, s) \);
44: while \( W \neq \emptyset \) do
45: \( e \leftarrow \text{SELECT FROM } W[/e] : src \sim \text{ dst} \)
46: if IsReachable(src, dst) then
47: \( \text{continue} \);
48: \( \text{foreach } o \in pts(src) \)
49: // \( L \) : a set of nodes whose points-to sets may change
50: \( L \leftarrow L \cup \text{ dst} \);
51: while \( L \neq \emptyset \) do
52: \( v \leftarrow \text{SELECT FROM } L \)
53: if IsReachable(o, v) then
54: \( \text{ continue} \);
55: \( pts(v) \leftarrow pts(v) \setminus o \);
56: \( \text{foreach } (v \sim \text{ dst}) \in E_p \)
57: \( L \leftarrow L \cup \text{ dst} \);
58: // process complex statements
59: \( \text{foreach } (C.m, s) \in CS(dst) \)
60: \( W \leftarrow W \cup FINDEDGES(C.m, s) \);

Algorithm 3 FINDEDGES(C.m, s)
1: Output: \( E \) – a set of edges, initially empty.
2: Notation: \( \bigcup \) – union for addition;
3: removal for deletion.
4: switch \( s \) do
5: case 1: \( x = \text{new } C; E \leftarrow E \cup O.f \sim \text{ x} \);
6: case 2: \( x = y; E \leftarrow E \cup y \sim \text{ x} \);
7: case 3: \( x = y.f \):
8: \( \text{foreach } O \in pts(y) \)
9: \( E \leftarrow E \cup O.f \sim \text{ y} \);
10: \( CS \leftarrow CS \bigcup (y, (C.m, s)) \);
11: case 4: \( x \sim \text{ y.f} \):
12: \( \text{foreach } O \in pts(x) \)
13: \( E \leftarrow E \cup y \sim O.f \);
14: \( CS \leftarrow CS \bigcup (x, (C.m, s)) \);
15: case 5: \( x = o.m'(y') \):
16: \( \text{foreach } O \cup \in pts(o) \)
17: if \( C.m \sim C'.m' \in \bigcup \text{ N} \) then
18: \( \text{foreach } s' \in C'.m'(y') \{ s' \text{ return } z \} \)
19: \( E \leftarrow E \cup \text{ FINDEDGES}(C.m', s') \);
20: \( E \leftarrow E \cup (y \sim y', z \sim x) \);
21: \( E \leftarrow E \bigcup (C.m \sim O.f) \);
22: \( CS \leftarrow CS \bigcup (o, (C.m, s)) \);
23: return \( E \).

Figure 4: An example of edge deletion.

to set of \( y \). Our new algorithm (shown in Algorithm 2 lines 41-59) hence lazily updates the points-to-set by checking the path reachability beforehand, using the function IsReachable(x,y). The algorithm updates the points-to-set of \( y \) (denoted by pts(y)) only when IsReachable(x,y) returns false and it removes an abstract object \( O \) from pts(y) only when IsReachable(O,y) returns false. Consider the example in Figure 4 where the edge \( x \sim y \) is deleted. We first check if \( x \) is still reachable to \( y \), if yes (e.g., in the existence of the path \( x \sim p \sim y \)), we simply stop. If not, we go on to check the reachability from each abstract object \( o \) in pts(x) to \( y \). If \( o \) is not reachable to \( y \), we remove \( o \) from pts(y) and continue to check the reachability from \( o \) to the nodes that are reachable from \( y \) (e.g., \( z \) in the example) and propagate the removal if not reachable. Otherwise, \( o \) remains in pts(y) and we stop the propagation for \( o \). If the path \( x \sim p \sim y \) does not exist, after deleting \( x \sim y \), \( o_1 \) can no longer reach \( y \) and \( o_2 \) is hence removed from pts(y). The removal is propagated to pts(z) because \( z \) is reachable from \( y \) and \( o_2 \) is no longer reachable to \( z \). However, \( o_2 \) is still reachable to \( y \) via the path \( o_2 \sim w \sim q \sim y \) and \( o_2 \) remains in pts(y). The computational improvement here is that we can skip the propagation to pts(z), because pts(y) is unchanged.

3.3 Soundness and Other Optimizations

Soundness. Both of our change-aware happens-before graph construction and points-to analysis algorithms are sound. Our incremental computation on the happens-before graph and points-to graph produces the same state as the one-shot whole-program computation. The key to the proof is that adding/removing a statement computed by our algo-
The size of the smaller benchmarks ranges from 68 to 2.8K lines of code. The size of real-world programs typically ranges from 10K to 172K. All benchmarks except H2 are self-contained, i.e., each of them has a single entry point (the main method) for the analysis. For H2, we use the test driver class org.h2.test.synth.thread.TestMulti in the H2 test suite as the entry point. All JDK libraries (e.g., java.lang.* and java.util.*) are included in the analysis except those excluded by WALA by default, such as java.awt.* and sun.*. These JDK libraries account for the majority of the classes (the number reported in Column 3) for building the class hierarchy. Nevertheless, for points-to analysis, only those methods that are reachable from the entry points are analyzed. The number of reachable methods for these benchmarks ranges from 927 to 3.7K, containing 27K to 108K SSA instructions. The points-to graph for each benchmark contains 9K-41K pointer keys (i.e., reference variables), 1.2K-5K instance keys (i.e., object allocation sites), and 60K-2.5M points-to edges. For WebLech, its points-to graph is much larger and more dense than the other benchmarks. It is particularly interesting for our evaluation of scalability because Anderson’s algorithm in the worst case is cubic in the size of the graph.

**Evaluation Methodology.** ECHO can be used by developers starting either from an empty project or from an existing code base. We choose the latter scenario for our evaluation. For each benchmark, we run the experiment in three phases: (1) run points-to analysis and static happens-before graph construction for the whole program; (2) delete a statement and run race detection; (3) add the deleted statement back and run race detection. Phase 1 is needed only once for ECHO. Phases 2 and 3 are performed for each statement in each method.

To understand the performance improvement of our novel change-aware algorithms, we also implement a whole program race detector and compare its performance with ECHO. The whole program race detector uses the same hybrid algorithm and the same pointer analysis as we use in ECHO. The only difference is that for every change it has to re-run the whole-program analysis. Hence, the whole program race detector and ECHO have the same race detection ability (i.e., report the same races with same precision and recall), except that ECHO is faster.

In addition, to show the improvement of our reachability-based algorithm for handling deletion, we compare its performance with the reset-and-recover algorithm.

All experiments were performed on an Apple Mac Pro with 2.5GHz dual-core Intel i5 processor and 4GB of memory running Java HotSpot 64-bit Server VM version 1.8.0.

### 4.1 Performance

Table 4 reports the performance of ECHO compared to the whole program race detector. Column 2 reports the time for performing the points-to analysis for each benchmark. The points-to analysis time typically ranges between 2s-5s, except for WebLech, which takes 16.4s because of its large points-to graph. Column 3 reports the time taken by the whole program race detector to detect races upon a change. The whole program race detector needs 2.5s-16.5s to detect races. Columns 4-5 report the average and worst-case time (including all incremental analyses and the race detection) taken by ECHO for each change.

For most benchmarks (including H2 and Pool), ECHO takes less than 1ms on average per change and 1s in the worst case. Compared to the whole program race detector, ECHO is typically four orders of magnitude faster. The only exception is WebLech, which has a much larger and more dense points-to graph. However, even for WebLech,
real-world systems. In particular, for the average case and 2.3X faster for the reset-then-recover algorithm takes 0.3–54s. On average, the reachability-based algorithm takes 0.46–368ms. In the worst case, the reachability-based and the reset-then-recover algorithms for handling statement deletions. In the average case, the reachability-based algorithm takes 0.4–8.9ms per change, whereas the reset-then-recover algorithm takes only 5ms on average for each change and 6.4s in the worst case, which is 41X and 8.5X faster.

Addition & Deletion. Table 5 compares the performance between addition and deletion. Columns 2-3 and Columns 4-5 report the average and worst case time, respectively, taken by ECHO for adding and deleting a statement. Columns 6-7 report the percentage of instructions that take over 91% of all the statements take less than 0.1s to handle. Overall, addition (0.06–1.1ms on average and 6.4s in the worst case) is much faster than deletion (0.4–9ms on average and 21–219ms in the worst case). The reason is that addition does not involve the complex invalidation of existing points-to sets. Nevertheless, over 91% of all the statements take less than 0.1s.

Reachability-based Deletion vs Reset-then-recover. Table 6 compares the performance between the reachability-based and the reset-then-recover algorithms for handling statement deletions. In the average case, the reachability-based algorithm takes 0.4–8.9ms per change, whereas the reset-then-recover algorithm takes 0.46–368ms. In the worst case, the reachability-based algorithm takes 0.15–6.4s, while the reset-then-recover algorithm takes 3.3–54s. On average, the reachability-based algorithm is 4.6X faster than reset-then-recover for the average case and 2.3X faster for the worst case. The speedup is more significant for the real-world systems. In particular, for WebLech, the reset-then-recover algorithm takes 368ms on average and 54.2s in the worst case, whereas the reachability-based algorithm takes only 8.9ms and 6.4s, respectively, which is 41X and 8.5X faster.

4.2 Recall & Precision

Table 7 reports the results of detected data races by ECHO. Each race has a unique signature, i.e., a pair of program statements. For several benchmarks, they have one or more known data races. We first studied all the known races and manually inspected the races reported by ECHO. We found that ECHO reported all those known races in these benchmarks. ECHO detected 23 true races and 13 false positives in total (all these races are available at [13]). The precision is 64% (23/36) and recall 100%. Nevertheless, ECHO may still miss certain true races in other Java programs because of its limited support of language features (e.g., ECHO does not handle reflection) and its use of may-alias (as described in Section 3.3). However, both of these two issues are fundamental to static analysis.

False Positives. ECHO reported 13 false positives in these benchmarks. The false positive ratio is 36% (13/36). Previous research [12, 1] has shown numerous sources of false positives raised by static analyses. We identified three main

### Table 3: Benchmarks. JDK libraries are also analyzed except those excluded by WALA by default.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>#Classes</th>
<th>#Methods</th>
<th>#SSAInstructions</th>
<th>#PointerKeys</th>
<th>#InstanceKeys</th>
<th>#PointerEdges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>68</td>
<td>7091</td>
<td>927</td>
<td>26881</td>
<td>8832</td>
<td>1178</td>
<td>58540</td>
</tr>
<tr>
<td>HttpClient</td>
<td>256</td>
<td>7084</td>
<td>1569</td>
<td>52681</td>
<td>17044</td>
<td>2925</td>
<td>32463</td>
</tr>
<tr>
<td>Manager</td>
<td>171</td>
<td>7090</td>
<td>956</td>
<td>27811</td>
<td>9121</td>
<td>1221</td>
<td>67671</td>
</tr>
<tr>
<td>MergeSort</td>
<td>298</td>
<td>7089</td>
<td>971</td>
<td>28522</td>
<td>9401</td>
<td>1275</td>
<td>66994</td>
</tr>
<tr>
<td>Racyce</td>
<td>294</td>
<td>7091</td>
<td>930</td>
<td>27447</td>
<td>8859</td>
<td>1179</td>
<td>58932</td>
</tr>
</tbody>
</table>

**Table 5: Performance of Addition and Detection.**

<table>
<thead>
<tr>
<th></th>
<th>Fast-Insts: instructions that take &lt;0.1s to handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
</tr>
<tr>
<td></td>
<td>delete</td>
</tr>
<tr>
<td>Example</td>
<td>0.4ms</td>
</tr>
<tr>
<td>FileWriter</td>
<td>1.7ms</td>
</tr>
<tr>
<td>Loader</td>
<td>0.44ms</td>
</tr>
<tr>
<td>Manager</td>
<td>0.5ms</td>
</tr>
<tr>
<td>MergeSort</td>
<td>0.53ms</td>
</tr>
<tr>
<td>Racyce</td>
<td>0.44ms</td>
</tr>
<tr>
<td>Pool</td>
<td>0.43ms</td>
</tr>
<tr>
<td>WebLech</td>
<td>8.9ms</td>
</tr>
<tr>
<td>H2</td>
<td>1.6ms</td>
</tr>
</tbody>
</table>

**Table 6: Performance of detection algorithms.**

<table>
<thead>
<tr>
<th></th>
<th>Reach-based</th>
<th>Reset-then-recover</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td>Example</td>
<td>0.4ms</td>
<td>1.9ms</td>
</tr>
<tr>
<td>FileWriter</td>
<td>1.7ms</td>
<td>2.4ms(1.4X)</td>
</tr>
<tr>
<td>Loader</td>
<td>0.44ms</td>
<td>1.97ms</td>
</tr>
<tr>
<td>Manager</td>
<td>0.5ms</td>
<td>2.69ms</td>
</tr>
<tr>
<td>MergeSort</td>
<td>0.53ms</td>
<td>2.38ms</td>
</tr>
<tr>
<td>Racyce</td>
<td>0.44ms</td>
<td>2.26ms</td>
</tr>
<tr>
<td>Pool</td>
<td>0.43ms</td>
<td>1.22ms</td>
</tr>
<tr>
<td>WebLech</td>
<td>8.9ms</td>
<td>6.4s</td>
</tr>
<tr>
<td>H2</td>
<td>1.6ms</td>
<td>1.08s</td>
</tr>
</tbody>
</table>

ECHO takes only 5ms on average for each change and 6.4s in the worst case, which is a 3000X speedup over the whole program race detector on average and 2.5X in the worst case.

Figure 5: False positives on “failureCount” in WebLech due to the lack of object sensitivity.

```java
class Spider run()
{
    class URLGetter getURL()
    {
        class Spider run()
        {
            if(failureCount > 10)
            {
                failureCount = 0;
            }else
            {
                failureCount++;
            }
        }
    }else
    {
        failureCount++;
    }
}
```
false positives on “m_DataArray[x]” in FileWriter due to indistinguishable array indexing.

Figure 6: False positives on “m_DataArray[x]” in FileWriter due to indistinguishable array indexing.

sources in our experiments:

Object Sensitivity. Identifying static objects by their allocation sites is imprecise. It can often lead to false positives between accesses to objects allocated at the same program location but are different. Consider an example in WebLech (Figure 5). The program starts two concurrent threads, both executing the run method of the Spider class. Each thread creates a new URLGetter object at line 163 and uses it to fetch URL, which accesses the object field failureCount at lines 63, 67, 132. ECHO reported 6 false positives on failureCount because the underlying pointer analysis does not distinguish the two URLGetter objects allocated at the same site by the two threads.

Indistinguishable Array Indexing. ECHO does not distinguish between different array indexes, which can lead to false alarms because accesses to different elements of the same array are considered as to the same memory. This issue can be more complicated when the array indexes are under complex data flow and path conditions. Consider an example in FileWriter (Figure 6). Thread T1 writes to a shared array m_DataArray with index m_iCount at line 38. Thread T2 reads m_DataArray with index place, which is computed by a random value mod m_iCount. The two indexes cannot be equal because of the mod operation. ECHO reported 2 false positives in FileWriter due to this problem.

Ad Hoc Synchronization. ECHO handles standard thread synchronizations in Java such as thread fork, join and the synchronized keyword but does not recognize ad hoc synchronizations. Missing ad hoc synchronizations caused several false positives in our experiments. Due to space reasons we refer the readers to our technical report [27] for a detailed example of this issue.

These issues open several interesting directions that we plan to investigate in future work. For example, adding more object and context sensitivity [28, 29, 16] in the points-to analysis could reduce false positives. A more precise array index analysis [30, 31, 32] could alleviate the second issue, and recognizing ad hoc synchronizations [33, 34, 35] could further improve the precision of ECHO.

5. RELATED WORK

A large number of static race detection techniques have been proposed, including many types systems [36, 37, 38, 39, 40, 41, 42], scalable whole-program analyses [1, 2, 43, 44, 12], model checking [45, 46], and other specialized techniques [47, 48]. The key advantage of static race detection is that it provides the potential to detect all races over all program paths, which eliminates false negatives, although most techniques in practice sacrifice soundness for scalability.

A primary limitation of static analysis is that it is imprecise and may produce false positives. A few sophisticated data flow analyses [12, 43, 44] have been proposed to improve precision via more expensive analysis. Compared to existing techniques, ECHO amortizes the analysis cost across many small program changes and avoids redundant computation through change-aware analysis. Moreover, ECHO works in the IDE and can detect races and warn the developers to fix the races as they are introduced.

Type-based race checking systems [39, 42, 36, 37, 38] can perform well in the IDE, but they typically require a significant amount of manual annotations and/or work only for a specific language. ECHO is fully automatic without any annotation and works for a realistic Java-like language.

Praun et al. [49] propose an object use graph (OUG) model that statically approximates the happens-before relation between accesses to a specific object. The key differences between OUG and our SHB graph are that the SHB model that statically approximates the happens-before relation between accesses to a specific object. The key differences between OUG and our SHB graph are that the SHB graph is field-sensitive while OUG is object-sensitive and that OUG does not model lock operations.

Points-to analysis has been extensively researched [14, 28, 16, 29, 15] in several different dimensions, e.g., flow-sensitivity, context-sensitivity, heap modeling, etc. Precise points-to analysis is NP-hard [15]. Any practical points-to analysis must approximate the exact solution and balance between precision and performance.

A few incremental and demand-driven points-to analysis algorithms have been proposed, based on CFL reachability [50, 17], logic programming [51], and data flow analysis [52, 18]. However, demand-driven approaches do not handle changes, and existing incremental approaches cannot efficiently handle code deletion. Moreover, none of them has been applied in IDEs for multithreaded programs before.

6. CONCLUSION AND FUTURE WORK

We have presented a new IDE-based static race detection technique and a tool, ECHO, that can detect data races as soon as they are introduced into the program. ECHO is powered by a set of novel change-aware static analyses that efficiently compute change-relevant program information upon code changes without re-analyzing the whole program. Our results on a variety of multithreaded benchmarks and real-world Java applications show that ECHO can detect races within milliseconds upon a code change with a reasonable precision. In future work, we plan to conduct empirical studies with developers to evaluate the usability and usefulness of ECHO for diagnosing and fixing real races. We also plan to improve the precision of ECHO by addressing the sources of false positives as discussed in Section 4.2.

7. ACKNOWLEDGEMENTS

The authors would like to thank Lawrence Rauchwerger and the anonymous reviewers for their constructive comments on earlier versions of this paper. This research is supported by faculty start-up funds from Texas A&M University, a Google Faculty Award, and NSF award CCF-1552935.
8. REFERENCES


[35] Le Yin. Effectively recognize ad hoc synchronizations


